7
Digital Signatures and Their Verification

In this chapter, we will learn about **digital signatures** and how to sign content and verify the signature later. Digital signatures have many important practical applications, such as signing documents, certificates, software, and network requests and responses, including financial transactions. Digital signatures are also important for the security of cryptocurrencies, such as Bitcoin and others. Digital signature algorithms are a necessary building block for client and server identity verification in secure network protocols, such as *TLS*, *SSH*, and *IPsec*. Digital signatures are used both in the *TLS* protocol handshake and for signing **X.509 certificates**, which are used for identity presentation and verification in *TLS*. More information about X.509 certificates will be provided in *Chapter 8*, *X.509 Certificates and PKI*. The TLS protocol will be covered in *Part 4*, *TLS Connections and Secure Communication*. Digital signatures are also used in secure messaging standards, such as **Pretty Good Privacy** (**PGP**) and **Secure Multipurpose Mail Extensions** (**S/MIME**). For example, OpenSSL releases are signed by *PGP* digital signatures.

We will also provide an overview of the digital signature algorithms that are supported by OpenSSL and recommendations on which digital signature schemes to use and which to avoid. In the practical part of this chapter, we will learn how to digitally sign content and verify the produced signature on the command line and programmatically using C code.

In this chapter, we are going to cover the following topics:

- Understanding digital signatures
- Overview of digital signature algorithms supported by OpenSSL
- How to generate an elliptic curve keypair
- How to sign and verify a signature on the command line
- How to sign programmatically
- How to verify a signature programmatically

Technical requirements

This chapter will contain commands that you can run on a command line and C source code that you can build and run. For the command-line commands, you will need the `openssl` command-line tool, as well as some OpenSSL dynamic libraries. To build the C code, you will need OpenSSL dynamic or static libraries, library headers, a C compiler, and a linker.

We will implement some example programs in this chapter, in order to practice what we are learning. The full source code of those programs can be found here: `https://github.com/PacktPublishing/Demystifying-Cryptography-with-OpenSSL-3/tree/main/Chapter07`

Understanding digital signatures

A digital signature is an array of bits that provides cryptographically strong guarantees of authenticity, integrity, and non-repudiation of a digital message. What do those guarantees mean? Let's take a look:

- **Authenticity** means that the message is coming from the claimed sender, provided that only the claimed sender possesses the private key that was used to produce the signature.
- **Integrity** means that the message has not been changed by a third party during transmission, for example.
- **Non-repudiation** means that the sender cannot deny that they produced the signature, provided that no one else has had access to the private key that was used to produce the signature.

A digital signature is produced using a *private key* that can be verified using the corresponding *public key*. Hence, digital signature and verification algorithms are considered **asymmetric cryptography algorithms**, even though they are not **asymmetric encryption algorithms**.

It is important to know that when a message is signed, usually, the digital signature algorithm is not applied to the message itself. Instead, the signature algorithm is applied to the *message digest*, which is produced by some cryptographic hash functions, such as SHA-256. Cryptographers say that such signature schemes are based on the **hash-and-sign** paradigm.

A notable and arguable exception is the EdDSA signature algorithm. It's notable because it is the only popular signature algorithm that takes the whole message instead of its digest as the input. It's arguable because EdDSA internally hashes the input message anyway, albeit in a more sophisticated manner than hashing is done for other signature algorithms.

So, why is a message digest signed instead of the message itself? The reasons are the same as why a session key is used for asymmetric encryption. Asymmetric crypto algorithms are relatively slow – much slower than cryptographic hash functions. A fast hashing algorithm transforms a potentially very long input message into a short message digest of known length so that a slow digital signature algorithm would not need to process a lot of input. Such separation of work improves performance, compared to if the whole long message needs to be processed by the signature algorithm. Another reason is that a digital signature algorithm can only sign a limited volume of data in one block. It is inconvenient to split the message into several blocks with several signatures and invent a secure method of chaining the produced signatures and resisting various order-changing attacks.

Note how *effective and performant* asymmetric cryptography builds on top of symmetric cryptography. Asymmetric encryption uses symmetric session keys, while digital signatures use message digests. Therefore, it is important to learn about *symmetric* cryptography before *asymmetric* cryptography.

Difference between digital signatures and MACs

As we can see, digital signatures have something in common with **Message Authentication Codes** (**MACs**). Both digital signatures and MACs can provide authentication and integrity. However, there are some important differences:

- Digital signatures are produced using asymmetric private keys and verified using public keys. MACs use the same symmetric key to produce an authentication tag and its verification.
- It is possible to verify a digital signature without the signer giving away its secret private key. Anyone possessing the public key can verify the signature. With a MAC, the same secret key must be possessed by both the signer and the verifier. Only the verifier can verify that the message is coming from the claimed sender and only if the verifier did not authenticate the same message. A third party cannot verify the message. If there are several verifiers, it becomes even more difficult to verify the origin of the message, not only because of the claimed sender but also because any verifier possessing the same secret key could authenticate the message.
- Unlike digital signatures, MACs do not provide non-repudiation. For any message that the sender authenticated, they can claim that it was the verifier who authenticated it.

Now that we understand what a digital signature is, let's find out what kind of digital signature algorithms are provided by OpenSSL.

Overview of digital signature algorithms supported by OpenSSL

In this section, we will review the digital signature algorithms that are supported by OpenSSL and give some recommendations on which algorithms to use.

Reviewing RSA

The **Rivest-Shamir-Adleman** (**RSA**) algorithm was invented by Ronald Rivest, Adi Shamir, and Leonard Adleman. Ronald Rivest is the same person who invented symmetric encryption algorithms from the **RC** family and message digest algorithms from the **MD** family. The RSA algorithm was first published in 1977. It was the first popular asymmetric crypto algorithm.

RSA is quite a universal algorithm since it can both encrypt and sign messages. RSA is the slowest algorithm to sign but the fastest to verify for the same security level among the digital signature algorithms provided by OpenSSL.

The speed of both signing and verification depends on the key size. The signing speed quickly drops with the growth of the key size. The verification speed also drops, but not as quickly. Verification is 30-70 times faster than signing, depending on the key size.

RSA produces the largest signatures. An RSA signature is the same size as the key size that was used to produce the signature. For example, a signature that was produced by a 4,096-bit key will have a size of 4,096 bits or 512 bytes. Signatures produced by other algorithms are significantly smaller.

Another disadvantage of RSA is that RSA keys are much longer than keys of more modern **Elliptic Curve** (**EC**)-based algorithms. Also, to gain just a few bits of security, the size of an RSA key must be increased substantially.

RSA is quite an old algorithm. OpenSSL supports another old algorithm, DSA.

Reviewing DSA

The **Digital Signature Algorithm** (**DSA**) was invented by David W. Kravitz while he was working for the **National Security Agency** (**NSA**) of the United States. The DSA was first published in 1991 when NIST proposed the DSA for use in their **Digital Signature Standard** (**DSS**).

The security of the DSA relies on the computational difficulty of the discrete logarithm problem. Like RSA, the DSA uses long keys, such as 2,048-bit keys. As with RSA, the security level of a DSA key is much lower than its length. DSA keys happen to have the same security level as RSA keys of the same lengths. Hence, a table that maps RSA key length to its security level, as shown in *Chapter 6*, *Asymmetric Encryption and Decryption*, can also be used to find out DSA key security. For example, a 2,048-bit DSA key would have a security level of 112 bits.

The first DSS, adopted as FIPS-186 in 1994, only allowed DSA to be used with the SHA-1 hash function and key lengths up to 1,024 bits. The latest DSS, FIPS 186-4 from 2013, also allows us to use DSA with the SHA-224 and SHA-256 hash functions and key lengths up to 3,072 bits. OpenSSL supports DSA via all three aforementioned hash functions and keys with very long lengths. In my opinion, however, you should not need a DSA key that's longer than 4,096 bits.

Even though DSA keys are long, like RSA keys, they produce a signature much shorter than the key length. A DSA's signature length, in bits, is *approximately* `4*K_bits`, where `K_bits` is the security strength of the used DSA key. For example, a 4,096-bit RSA key produces a 512-byte signature, but a DSA key of the same length produces a 70- or 71-byte signature.

Currently, the DSA is not a very popular algorithm and is being superseded by ECDSA.

Reviewing ECDSA

The **Elliptic Curve Digital Signature Algorithm** (**ECDSA**) is a variant of the DSA algorithm that uses **Elliptic Curve Cryptography** (**ECC**). ECC is a form of public key cryptography that is based on the algebraic structure of elliptic curves over finite fields. ECC was invented by Neal Koblitz and Victor Miller in 1985. ECDSA was proposed by Scott Vanstone in 1992 as a comment on the first DSS standard.

ECDSA's advantage over DSA is that it has a shorter key size for the same security level. The security level of an elliptic curve key is approximately half of the key length. For example, a 224-bit key has a 112-bit security level. For comparison, a DSA key with the same security level must be as long as 2,048 bits. As with DSA, the ECDSA signature length, in bits, is *approximately* `4*K_bits`, where `K_bits` is the security strength of the used ECDSA key.

When you're generating an ECDSA key, you have to choose a curve. A curve has a name and defines how long an EC key will be when the key is generated based on that curve. For instance, a key generated based on the NIST P-256 curve will have a length of 256 bits. OpenSSL supports NIST curves and Brainpool curves. NIST curves are developed by NSA and standardized by NIST. Brainpool curves are proposed by the Brainpool workgroup, a group of cryptographers that were dissatisfied with NIST curves because NIST curves were not verifiably randomly generated, so they may have intentionally or accidentally weak security. Brainpool curves are standardized and used by the Federal Office for Information Security of Germany (Bundesamt für Sicherheit in der Informationstechnik, BSI). The Brainpool workgroup defined a method of verifiably randomly generating elliptic curves and claimed that the workgroup generated Brainpool curves using that method. However, a research group led by Daniel J. Bernstein tried to verify the Brainpool workgroup method and could not generate the same Brainpool curves using the described method. Bernstein's group later proposed their own elliptic curves and own EC signature algorithm called EdDSA. More details on EdDSA curves will be provided in the next section.

Despite these doubts, NIST curves are currently the most popular curves for ECDSA. The NIST curve suite includes two very fast curves: the P-256 curve and the P-224 curve. The NIST P-256 curve is 16 times faster than the Brainpool P256r1 curve when signing and five times faster when verifying a signature. The NIST P-224 curve, comparable in security to 2,048-bit DSA, is approximately five times faster than DSA-2048 when signing and two times faster when verifying. Other NIST curves have speeds comparable to Brainpool curves of similar security, meaning that sometimes, NIST curves are faster, while other times, Brainpool curves are faster.

It is important to mention that the standard implementation of ECDSA needs a good **Random Number Generator** (**RNG**) both for key generation and during the signing process. There is also an alternative implementation of ECDSA that does not need random data generation when signing that uses the hash of the private key instead of random data. A faulty RNG during signing may result in leaking the private key via the signature. Care should be taken while implementing ECDSA properly; otherwise, an implementation may be vulnerable to a timing attack that may also reveal the private key. OpenSSL once had an implementation that was vulnerable to a timing attack, but the vulnerability was fixed in 2011.

ECDSA is quite a popular algorithm, but it has a competitor: another EC-based signature algorithm, called EdDSA.

Reviewing EdDSA

Edwards-curve Digital Signature Algorithm (**EdDSA**) is another signature algorithm that uses elliptic curves. It was published in 2011 by a group of cryptographers led by Daniel J. Bernstein, the same person that developed the ChaCha symmetric stream cipher family and the Poly1305 MAC algorithm.

EdDSA is based on **twisted Edwards curves** and aims to be both faster and easier to implement securely than ECDSA. Unlike ECDSA, EdDSA does not need RNG during signing and thus cannot leak private keys because of bad RNG.

EdDSA supports two curves: the 253-bit Curve25519 and the 456-bit Curve448. Those curves are designed for good security, attempting to make it difficult to make implementation mistakes that may weaken security. The curves are also immune to timing attacks.

As with ECDSA, the security level of EdDSA signatures is half of the key length, meaning 126.5 bits for Curve25519 and 228 bits for Curve448. The signature length for Curve25519 is 64 bytes, while for Cruve448, it is 114 bytes.

Even though Curve25519 and Ed448 are designed for speed, they are not faster than the NIST P-256 curve. NIST P-256 is approximately two times faster than Curve25519, both for signing and signature verification. The NIST P-224 curve has about the same speed as Curve25519. However, Curve25519 is 8-13 times faster than other NIST and Brainpool curves of comparable security for signing, and 2-9 times faster than them for verification. Curve448 is approximately two times faster than NIST and Brainpool curves of comparable security for signing and 1.5-3 times faster for verification.

EdDSA is an unusual digital signature algorithm. Usual signature algorithms expect input in the form of a cryptographic hash of the input data. For clarity, the input data is the data to be signed. On the contrary, EdDSA expects the whole input data as input but allows the algorithm's user to provide a so-called prehash function that will be applied to the input data before further processing.

A prehash function can be an actual cryptographic hash function, such as SHA-256. Such a variant of EdDSA is called **HashEdDSA**. The HashEdDSA variant is the most similar to traditional signature algorithms because it feeds the input data hash to further processing. A prehash function can also be the identity function, the function that just returns its input. Such a variant of EdDSA is called **PureEdDSA**.

It's worth noting that the EdDSA algorithm internally uses the SHA-512 hash function when using Curve25519 and the SHAKE256 function when using Curve448. Hence, in the case of HashEdDSA, the input data will be hashed twice. Should PureEdDSA always be used, then? No, it's not that easy – please read on.

Another difference between EdDSA and the traditional signature algorithms is that EdDSA is a two-pass algorithm, which means it processes the input data twice. This means, in turn, that PureEdDSA does not support streamed input data, unlike traditional signature algorithms. If the input data is long and coming from slow storage or requested via an expensive network connection, PureEdDSA may not be the best choice.

OpenSSL 3.0 only supports the PureEdDSA variant of EdDSA and requires all the data to be put into memory for hashing. This can be problematic if the input data is very long, and memory is scarce. On the other hand, even though HashEdDSA is not directly supported by OpenSSL yet, the programmer can hash the input data themself first and then use PureEdDSA for further processing.

Now, let's look at the last signature algorithm we will review – SM2.

Reviewing SM2

Shang Mi 2 (**SM2**) is another elliptic curve-based digital signature algorithm. It is a national standard of the People's Republic of China. SM2 was invented by Xiaoyun Wang et al. and standardized by the Chinese Commercial Cryptography Administration Office in 2012.

SM2 only supports one curve: the 256-bit CurveSM2. CurveSM2's speed for signing and verifying is about the same as that of 256-bit Brainpool curves. The signature length for CurveSM2 is 71 bytes.

The SM2 signature algorithm is usually used with a 256-bit SM3 hash function. OpenSSL also supports combining SM2 only with SM3 in its high-level APIs.

As we can see, OpenSSL supports several digital signature algorithms. But which should you choose? Let's talk about it in the next subsection.

Which digital signature algorithm should you choose?

Recently, the EdDSA algorithm and Curve25519 have become quite popular. Curve25519 appears to be trusted by many security experts. If your data to be signed always fits into memory and you do not need compatibility with old software, choose EdDSA. It is worth mentioning that, at the time of this writing, major **Certificate Authorities** do not issue EdDSA-based X.509 certificates yet. Hence, if you need a TLS certificate for a server on the internet, you will need to choose a certificate that's been signed with another algorithm, for the time being.

If you want a more traditional signature algorithm or want to support streaming, then choose ECDSA, which is also very popular.

If you need interoperability with old software, or if you need very fast signature verification, then choose RSA.

With that, we have learned about different digital signature algorithms supported by OpenSSL and recommendations on which algorithm to use in which situation. That concludes the theoretical part of this chapter. The next section will be practical. There, we will learn how to generate an EC keypair and how to sign some data on the command line.

How to generate an elliptic curve keypair

I want to demonstrate the traditional approach to digital signatures when a message digest of the input data is signed. Hence, we will use ECDSA, not EdDSA, in our examples.

As we already know, a new keypair can be generated using the `openssl genpkey` subcommand. We will generate an EC keypair of the longest length available in OpenSSL, 570 bits, based on the NIST B-571 curve:

```
$ openssl genpkey \
    -algorithm EC \
    -pkeyopt ec_paramgen_curve:secp521r1 \
    -out ec_keypair.pem
```

Here, we have used the `-pkeyopt ec_paramgen_curve:secp521r1` switch to specify that we want to use the NIST B-571 curve. Which other curve names could be used instead of `secp521r1`? The full list of supported curves can be obtained using the following command:

```
$ openssl ecparam -list_curves
```

Once the keypair has been generated and written into the `ec_keypair.pem` file, we can inspect its structure:

```
$ openssl pkey -in ec_keypair.pem -noout -text
Private-Key: (570 bit)
priv:
    … hex values …
pub:
    … hex values …
ASN1 OID: sect571r1
NIST CURVE: B-571
```

As we can see, an EC keypair's structure is simpler than an RSA keypair's structure. It's just two long values – a private part and a public part.

Extracting the public key from the keypair can be done similarly to how we did it for RSA in *Chapter 6*, *Asymmetric Encryption and Decryption*:

```
$ openssl pkey \
    -in ec_keypair.pem \
    -pubout \
    -out ec_public_key.pem
```

Let's inspect the structure of the extracted public key:

```
$ openssl pkey -in ec_public_key.pem -pubin -noout -text
Public-Key: (570 bit)
pub:
    … hex values …
ASN1 OID: sect571r1
NIST CURVE: B-571
```

Here, we can see that the public key contains the same `pub` value as in the keypair and, naturally, does not contain the `priv` part.

To digitally *sign*, you need a *private key*, and to *verify* the signature, you need a *public key*. As a reminder, you should never share your private key or keypair. You should only distribute your public key.

As we explained in *Chapter 6*, *Asymmetric Encryption and Decryption*, when OpenSSL documentation mentions a private key, usually, it means a keypair. Also, there is no command-line command to extract the private part from a keypair. This can only be done programmatically by knowing how an EC key is constructed in the C code and picking up the right data. But extracting the private key from a keypair does not make much sense in practice because when OpenSSL needs a private key, it can use the whole keypair. If you need to sign something using OpenSSL, either on the command line or programmatically, just supply the whole keypair to OpenSSL – do not bother extracting the private key.

With that, we have generated our elliptic curve keypair. Now, let's sign something.

How to sign and verify a signature on the command line

OpenSSL provides several subcommands for signing and verifying signatures. Let's take a look:

- The deprecated RSA-specific `openssl rsautl` subcommand.
- The `openssl dgst` subcommand: This is usually used for message digest calculation but can also be used to sign the produced digests. This means that it cannot be used to sign PureEdDSA because that signature algorithm does not sign digests.
- The `openssl pkeyutl` subcommand: This subcommand can be used to sign with any signature algorithm supported by OpenSSL. Before OpenSSL 3.0, `openssl pkeyutl` did not support signing long inputs; the user had to make the message digest before signing. Since OpenSSL 3.0, `openssl pkeyutl` supports both "raw input," as it is called in the documentation, and a message digest as input.

We are going to use the `openssl pkeyutl` subcommand for our examples. Its documentation can be found on the `openssl-pkeyutl` man page:

```
man openssl-pkeyutl
```

Follow these steps to sign and verify a signature on the command line:

1. First, let's generate a file with sample data that we are going to sign:

   ```
   $ seq 20000 >somefile.txt
   ```

2. Now, let's sign the sample file using the SHA3-512 hash function and the EC keypair that we have already generated:

   ```
   $ openssl pkeyutl \
       -sign \
       -digest sha3-512 \
       -inkey ec_keypair.pem \
       -in somefile.txt \
       -rawin \
       -out somefile.txt.signature
   ```

3. Let's check the filesystem:

   ```
   $ cksum somefile.txt*
   3231941463 108894 somefile.txt
   2915754802 151 somefile.txt.signature
   ```

   The output of this operation is a 151 bytes long signature written to the `somefile.txt.signature` file.

4. Now, if we share the original `somefile.txt` file, the signature in `somefile.txt.signature`, and our public key in `ec_public_key.pem`, anyone can verify the signature. That's how it is done on the command line:

   ```
   $ openssl pkeyutl \
       -verify \
       -digest sha3-512 \
       -inkey ec_keypair.pem \
       -in somefile.txt \
       -rawin \
       -sigfile somefile.txt.signature
   Signature Verified Successfully
   ```

   Note that we used the `-sign` switch for signing and the `-verify` switch for signature verification.

5. What if the signed data was modified during transmission? Let's simulate such a situation by appending some data to the signed data and trying to verify it again:

   ```
   $ echo "additional data" >> somefile.txt
   $ openssl pkeyutl \
       -verify \
       -digest sha3-512 \
       -inkey ec_keypair.pem \
       -in somefile.txt \
       -rawin \
       -sigfile somefile.txt.signature
   Signature Verification Failure
   ```

   As we can see, the signature verification failed in this case. We have observed that if the signed data remains intact – that is, the signature verification succeeds, but the signed data has been altered – the signature verification procedure detects it and reports a failure.

Now that we've learned how to digitally sign and verify the produced signature on the command line, let's learn how to do it programmatically.

How to sign programmatically

OpenSSL 3.0 provides the following APIs for digital signatures:

- Legacy low-level APIs that consist of functions with algorithm-specific prefixes, such as `RSA_`, `DSA_`, and `ECDSA_`. These APIs have been deprecated since OpenSSL 3.0.
- The `EVP_PKEY` API: This API is not deprecated but is still low level. It is more convenient to use a high-level API.
- The `EVP_Sign` API: This API is high level but has some disadvantages. This API uses the key argument only after the whole input data has been read and hashed. Therefore, if there is a problem with the key, it will be discovered later rather than sooner. Another disadvantage is that this API is inflexible and does not allow you to set signing parameters to `PKEY_CTX` if the signature algorithm supports them. The OpenSSL documentation does not recommend this API.
- The `EVP_DigestSign` API: This is a high-level API where drawbacks of the `EVP_Sign` API have been fixed. The OpenSSL documentation recommends this API, so we are going to use it here.

We are going to develop a small `ec-sign` program that signs a file using the ECDSA algorithm, similar to how we signed with `openssl pkeyutl` in the previous section. Our program is going to be interoperable with `pkeyutl`, meaning that `pkeyutl` will be able to verify the signature that's produced by our program.

Here are some relevant manual pages for the API that we are going to use:

```
$ man PEM_read_PrivateKey
$ man PEM_read_PUBKEY
$ man EVP_DigestSignInit_ex
$ man EVP_DigestVerifyInit_ex
```

Our program will take three command-line arguments:

1. The name of the input file containing the data to be signed
2. The name of the output file where the signature will be written
3. The name of the file containing the signing keypair

Our high-level implementation plan will be as follows:

1. Load the keypair from the keypair file.
2. Create an `EVP_MD_CTX` object that will be used as the signing context.
3. Initialize the signing context with the hash function name and the loaded keypair.
4. Read the data to be signed chunk by chunk and feed it to the signing context.
5. Discover the signature length and allocate memory for the signature.
6. Finalize the signing and get the signature.
7. Write the signature into the output signature file.

Now it's time to implement our plan by writing the necessary code.

Implementing the ec-sign program

Let's get started with the implementation:

1. First, load the keypair:

   ```
   const char* pkey_fname = argv[3];
   FILE* pkey_file = fopen(pkey_fname, "rb");
   EVP_PKEY* pkey = PEM_read_PrivateKey(
       pkey_file, NULL, NULL, NULL);
   ```

2. Then, create the `EVP_MD_CTX` signing context and initialize it with the SHA3-512 hash function and the loaded keypair:

```
EVP_MD_CTX* md_ctx = EVP_MD_CTX_new();
EVP_DigestSignInit_ex(
        md_ctx,
        NULL,
        OSSL_DIGEST_NAME_SHA3_512,
        NULL,
        NULL,
        pkey,
        NULL);
```

3. Then, the signing context initialization is created. Let's feed the input data to the context:

```
const char* in_fname = argv[1];
FILE* in_file = fopen(in_fname, "rb");
while (!feof(in_file)) {
    size_t in_nbytes = fread(in_buf, 1, BUF_SIZE, in_
file);
    EVP_DigestSignUpdate(md_ctx, in_buf, in_nbytes);
}
```

   Once all the data has been fed to the context, we have to finalize the signing process and get the signature. But we don't know how much memory to allocate for the signature buffer!

4. Fortunately, the `EVP_DigestSignFinal()` function will give us the signature length if we supply `NULL` as the signature buffer pointer:

```
size_t sig_len = 0;
EVP_DigestSignFinal(md_ctx, NULL, &sig_len);
```

   Note that `EVP_DigestSignFinal()` is not the only OpenSSL function that can indicate the desired output buffer length this way. Many other OpenSSL functions will also write the wanted buffer length at the supplied length pointer if the supplied buffer pointer is `NULL`.

5. Now, we know the signature length and can allocate memory for the signature:

```
unsigned char* sig_buf = malloc(sig_len);
```

6. With that, we have a place to store the signature and can finally get it:

   ```
   EVP_DigestSignFinal(md_ctx, sig_buf, &sig_len);
   ```

   Note that this time, we supply `sig_buf` instead of `NULL` as the target signature buffer into the `EVP_DigestSignFinal()` function.

   Instead of using the `EVP_DigestSignInit_ex()`, `EVP_DigestSignUpdate()`, and `EVP_DigestSignFinal()` functions, we could use a one-shot function called `EVP_DigestSign()`, if all our data to be signed can be loaded into one contiguous buffer. We don't have to do it with ECDSA, but it would be the only way to make a PureEdDSA signature in OpenSSL 3.0.

7. Now that we have got the signature, let's write it into the output file:

   ```
   const char* sig_fname = argv[2];
   sig_file = fopen(sig_fname, "wb");
   fwrite(sig_buf, 1, sig_len, sig_file);
   ```

8. Finally, let's free buffers and objects to avoid memory leaks:

   ```
   free(sig_buf);
   free(in_buf);
   EVP_MD_CTX_free(md_ctx);
   EVP_PKEY_free(pkey);
   ```

The complete source code for our `ec-sign` program can be found on GitHub in the `ec-sign.c` file: `https://github.com/PacktPublishing/Demystifying-Cryptography-with-OpenSSL-3/blob/main/Chapter07/ec-sign.c`.

Running the ec-sign program

Let's run our `ec-sign` program and see how it works:

```
$ ./ec-sign \
    somefile.txt \
    somefile.txt.signature \
    ec_keypair.pem
Signing succeeded
```

The program has been reported as a success. Let's use `openssl pkeyutl` to check whether our program has produced a good signature:

```
$ openssl pkeyutl \
    -verify \
    -digest sha3-512 \
    -inkey ec_keypair.pem \
    -in somefile.txt \
    -rawin \
    -sigfile somefile.txt.signature
Signature Verified Successfully
```

It is very good that our program has been able to produce the correct signature. But what's even better is that we have learned how to write such a program. Our next adventure is to learn how to verify signatures programmatically.

How to verify a signature programmatically

To learn how to verify signatures programmatically, let's develop a small `ec-verify` program that can verify signatures produced by `ec-sign` or `openssl pkeyutl`.

Our program will take three command-line arguments:

1. The name of the input file containing the signed data
2. The name of the file containing the signature
3. The name of the file containing the verifying public key

Here is our high-level implementation plan for the program:

1. Load the signature.
2. Load the verifying public key.
3. Create an `EVP_MD_CTX` object that will be used as the verifying context.
4. Initialize the verifying context with the hash function's name and the loaded public key.
5. Read the signed data chunk by chunk and feed it to the verifying context.
6. Finalize the verification and find out whether the verification has succeeded or failed.

Now it's time to implement our plan.

Implementing the ec-verify program

Follow these steps for implementing the `ec-verify` program according to our plan:

1. First, we must load the signature. However, we don't know how much memory to allocate for it. So, let's open the signature file and find out its length:

```
const char* sig_fname = argv[2];
FILE* sig_file = fopen(sig_fname, "rb");
fseek(sig_file, 0, SEEK_END);
long sig_file_len = ftell(sig_file);
fseek(sig_file, 0, SEEK_SET);
```

   Such a seek-based method of finding out the file length may not be optimal, but it uses only the standard C library and thus is very cross-platform.

2. Now, when we know the signature length, we can allocate memory for the signature and load it from the signature file:

```
unsigned char* sig_buf = malloc(sig_file_len);
size_t sig_len = fread(sig_buf, 1, sig_file_len, sig_
file);
```

3. Next, load the verifying public key:

```
const char* pkey_fname = argv[3];
FILE* pkey_file = fopen(pkey_fname, "rb");
EVP_PKEY* pkey = PEM_read_PUBKEY(
    pkey_file, NULL, NULL, NULL);
```

4. Then, create the verifying context and initialize it with the SHA3-512 hash function and the loaded public key:

```
EVP_MD_CTX* md_ctx = EVP_MD_CTX_new();
EVP_DigestVerifyInit_ex(
    md_ctx,
    NULL,
    OSSL_DIGEST_NAME_SHA3_512,
    NULL,
    NULL,
    pkey,
    NULL);
```

5. Now that the verifying context has been initialized, let's feed it with the signed data:

```
while (!feof(in_file)) {
    size_t in_nbytes = fread(in_buf, 1, BUF_SIZE, in_
file);
    EVP_DigestVerifyUpdate(md_ctx, in_buf, in_nbytes);
}
```

6. Once all the signed data has been processed by the context, we can find out the verification status:

```
int status = EVP_DigestVerifyFinal(md_ctx, sig_buf, sig_
len);
```

   The `EVP_DigestVerifyFinal()` function returns `1` if the signature verification is successful. Any other returned value indicates a failure.

   Like the signing API, the verifying API also has a one-shot verification function called `EVP_DigestVerify()` that does not take data chunk by chunk but requires the whole signed data to be placed in a contiguous memory block. Using `EVP_DigestVerify()` is, as of OpenSSL 3.0, the only way to verify signatures for algorithms that do not support streaming, such as PureEdDSA.

7. Once we have got the verification status from the `EVP_DigestVerifyFinal()` function, we don't need our used buffers and objects anymore and can free them:

```
free(in_buf);
EVP_MD_CTX_free(md_ctx);
EVP_PKEY_free(pkey);
free(sig_buf);
```

The complete source code for our `ec-verify` program can be found on GitHub in the `ec-verify.c` file: `https://github.com/PacktPublishing/Demystifying-Cryptography-with-OpenSSL-3/blob/main/Chapter07/ec-verify.c`.

Running the ec-verify program

Let's run our `ec-verify` program and try to verify the signature that's been produced by the `ec-sign` program:

```
$ ./ec-verify \
    somefile.txt \
    somefile.txt.signature \
    ec_public_key.pem
Verification succeeded
```

Good – our `ec-verify` program has successfully verified a valid signature.

Let's try to change the signed data and check whether verification will fail:

```
$ echo "additional data" >> somefile.txt
$ ./ec-verify \
    somefile.txt \
    somefile.txt.signature ec_public_key.pem
EVP_API error
Verification failed
```

As we can see, our `ec-verify` program can distinguish a valid signature from an invalid one.

This section concludes the practical part of this chapter. Let's proceed to the summary.

Summary

In this chapter, we have learned about the concept of digital signatures, how they use message digest, and how digital signatures differ from message authentication codes. We also learned about the cryptographical guarantees that digital signatures provide. Then, we reviewed the digital signature algorithms supported by OpenSSL, discussed their technical merits, and mentioned a little bit of their history. We finished the theoretical part with recommendations on which digital signature algorithm to choose in which situation.

In the practical part, we learned how to sign with ECDSA and verify signatures on the command line. Then, we learned how to sign and verify signatures programmatically using C code.

In the next chapter, we will learn about X.509 certificates and **Public Key Infrastructure** (**PKI**) based on X.509 certificates.